A HACKER'S ASSISTANT

Document Date: 10/22/08

1 Introduction

Aha! is a program of the superoptimizer type that can be used to find short branch-free code sequences for certain simple functions such as

- the absolute value function,
- a comparison predicate such as $x \leq 0$, and
- swapping the two low-order bits of a register.

It works by exhaustive search over programs of a given length. As a practical matter the length is limited to four instructions for most machines.

The user supplies a C program that computes the desired function. The function must do a calculation on one or two integer arguments. It cannot have any "memory" and should not have side effects. The code supplied can be inefficient because it is executed only a few times. As an example, below is the user-supplied function for computing the absolute value of an integer.

```
int userfun(int x) {
   if (x >= 0) return x;
   else return -x;
}
```

The user also defines the target machine by supplying a definition of each computational instruction in the machine's instruction set, and a subroutine that performs its function. The instructions must operate only on data in registers and have one output register operand and a maximum of three input register operands. (Some of the registers hold constants and thus serve as "immediate" operands.) Side effects such as a carry, overflow, and a condition register setting cannot be handled with this program. The program handles only computational instructions; it does not handle memory operations (loads, stores, and so on) or branches.

As examples, below are the definitions of four instructions that the machine might have. The meaning of the fields will be described in more detail later, but briefly, they are: name of subroutine to simulate the instruction, number of operands, commutative indicator, starting register number for operands, name for printing the instruction in assembly language style, and two character strings used for printing the discovered code in algebraic style.

```
{add, 2, 1, {RX, 2, 0}, "add", "(",    " + "  },
{sub, 2, 0, { 2, 2, 0}, "sub", "(",    " - "  },
{mul, 2, 1, {RX, 3, 0}, "mul", "(",    "*"    },
{pop, 1, 0, {RX, 0, 0}, "pop", "pop(", ""     },
```

Below is an example of a subroutine for simulating an instruction.

```
int add(int x, int y, int) {return x + y;}
```

Most of the instruction simulation routines are simple one-liners like this, but a few, such as for *population count*, are several lines of code.

After supplying the user function and the machine description (which in many cases would be a small modification of the one supplied), the user compiles Aha! with a simple "make" command. It is always completely recompiled after a change to either the user function or the machine description; on a modern machine this takes only two or three seconds.

Finally, the program is executed by typing in its name followed by an integer that specifies the length (number of instructions) of the programs to be tried. (It does not try programs of length 1, then 2, and so on.) Aha! displays each program as it is found, and also writes the display output to an output file. As an example, for the absolute value function on a typical RISC machine, one of the solutions it finds is displayed as shown below.

```
shrs  r1,rx,30
or    r2,r1,1
mul   r3,r2,rx
Expr: (((x >>s 30) | 1)*x)
```

First are shown the three instructions in assembly language style (*shift right signed* 30 positions, *or* with the immediate value 1, and *multiply*; the input argument is in register `rx` and the target register is on the left). The assembly language code is followed by a formula for the function in algebraic notation (`>>s` is *shift right signed*, `|` is *or*, and `*` is *multiply*).

This run of the program, incidentally, took 0.33 seconds of process time on a 667 MHz Pentium III workstation. 1.86 million instructions were simulated, from which we calculate that the program used 118 machine cycles per evaluation. The wall clock time, which includes the time to start and end the program, is only a fraction of a second longer than the process time, assuming there is not much competition for machine cycles when Aha! is running.

As another example, consider the problem mentioned above of interchanging the rightmost (least significant) bits of a register. The straightforward code:

```
(x & 0xfffffffc) | ((x & 1) << 1) | ((x & 2) >> 1)
```

uses seven instructions plus, for some machines, a load immediate of the large constant. A method is given in *Hacker's Delight* for doing it in six instructions (bottom of page 40, with $m = 1$).

This problem was given to Aha! as set up for a basic RISC machine with six nonzero immediate values and four immediate shift values (1, 2, 30, and 31). For programs of length three, no solutions were found. For programs of length four, Aha! ground away for 101 seconds (667 mHz machine) and came up with 20 solutions. Many were trivial variations of one another, and 12 of the solutions used *divide* (signed).

One of Aha!'s solutions that does not use *divide* is shown below.

```
add   r1,rx,3
and   r2,r1,2
shr   r3,3,r2
xor   r4,r3,rx
Expr: ((3 >>u ((x + 3) & 2)) ^ x)
```

In truth, this is not really a 4-instruction solution on most computers, because most computers do not allow an immediate value for the first operand of a shift instruction. But it's close; in some situations the required *load immediate* of 3 could be moved out of a loop, leaving four instructions for the function in the loop. If Aha! were set up to not use an immediate value in this position (which is easy to do), the above solution would be missed, unless the user is willing to wait several hours for a search over programs of length five.

Seven uninteresting variations of the above were found, such as *using shift right signed* instead of *shift right*, and computing $-2-x$ instead of $x+3$ by the first instruction (which are equivalent for the purposes of this code).

Below is a solution that uses *divide* but not with immediate operands.

```
shl   r1,rx,30
sub   r2,r1,1
div   r3,r1,r2
add   r4,r3,rx
Expr: (((x << 30)/((x << 30) - 1)) + x)
```

How long would it take to find 5-instruction solutions? The jump from three to four instructions increased the execution time by a factor of about 306 (101/0.33). The jump from four to five would increase execution time by a factor somewhat larger than this (because each last instruction must try one more register for its input operands), probably 350 or so. This gives an estimated execution time of 35,350 seconds, or about 10 hours.

Aha! is based on an idea by Henry Massalin [HM]. His program has been generalized and made widely available by Torbjörn Granlund and Richard Kenner as the GNU Superoptimizer [GK]. Like its predecessors, Aha! is oriented toward a machine with a number of 32-bit general purpose registers, and it deals with integer and logical instructions only.

Aha! differs from its predecessors in several details. Basically, it is more algebraically oriented. As provided, the instruction set consists of the usual computational instructions found on most computers (*add*, *and*, *shift left*, and so on). But (unlike most computers) any operand can be either a 32-bit immediate value

or a register. If this generality is not wanted, it is a simple matter to specify that certain operands of certain instructions cannot be immediate values.

Aha! is not perfect. It can miss solutions, mainly because it does not try all immediate values. It can print out multiple solutions when they are only trivially different, for example the solution given above for the *absolute value* function is followed by the same solution except with the 30 changed to 31. As provided, Aha! can give an *n*-instruction solution that cannot really be implemented in *n* instructions because it uses an immediate value for an operand that the machine in mind does not allow. To solve a difficult problem the user may have to tinker with the instruction set and the set of immediate values used. For example, to find efficient code for multiplying by 5, the *multiply* instruction should be deleted from the instruction set (by simply commenting-out one line in a header file). There are other shortcomings, discussed in a later section. The program should be viewed not as an infallible oracle but rather as a hardworking assistant.

2 User's Guide

Aha! was developed and is maintained with the Windows 2000 and Windows XP operating systems, using the Cygwin GNU C compiler. The user must have this compiler installed. A Windows version is available at no charge from http://cygwin.com/ (click "Install Cygwin now").

Calin Cascaval has kindly made a Makefile for Aha! on Red Hat Linux with gcc 3.2.1, and finds that Aha! runs successfully on that system. This Makefile and Aha! may very well run successfully on other Unix-like systems. However, the user is cautioned that Aha! is seldom tested on Linux or any other operating system except Windows.

Any interested party is invited to port Aha! to other platforms, but the author has no plans to do so. It should be easy to port, because it is not a large program (less than 700 lines of code, including the machine description for a basic RISC), it uses straightforward ANSI C (although with probably a few GNU extensions), and it has no assembly language or calls on unusual operating system services. Aha! is "freeware." It is free of charge, and you are free, and in fact encouraged, to modify it and distribute it in any way you like. This program is made available to you without any warranty, either expressed or implied. The author makes no claim that the code is free of errors; in fact it is likely to have some bugs. You may charge a fee for modifying and distributing this code, but you may not restrict further modifying and distributing of your derivative work by any means, such as by licensing or patenting.

This description assumes that you, the user of Aha!, are reasonably skilled in the use of a text editor and in using Windows from a Command Prompt window.

Aha! consists of the following source files:

```
xxx.h
aha.c
```

where "xxx" is a name of your choice. There is also a file named `make.bat` for compiling and linking the program on Windows, and `Makefile` for compiling and linking on Linux. There are a few other files for documentation and miscellaneous functions that would not often be used. To perform experiments with Aha!, you modify only the first file, shown as xxx.h (unless of course you wish to alter the program to do new things).

First you supply C code that defines the function that you wish to implement more efficiently. This would usually be a very simple program and it is located near the beginning of xxx.h. An example is shown on page 1.

Next you define the instruction set and other parameters of the computer for which you desire to find efficient code (the "target machine"). These definitions are all contained in a single header file; `abs.h` and `avg.h` are included as examples. You can modify either `abs.h` or `avg.h`, and save the new file under a name of your choice.

Modifying the Header File

The example header files `abs.h` and `avg.h` contain quite a few comments so it should not often be necessary to refer to this documentation. Below is a description of the data you might want to modify.

`NARGS` The number of arguments of `userfun` (1 or 2).

`debug` 1 for debugging output, 0 for none.

`counters` 1 to count and display the number of instruction evaluations, 0 to not do this (using 0 does not speed execution significantly).

`NBSM` Mask applied to shift amounts. Use 63 for mod 64 shifts, and 31 for mod 32 shifts.

`trialx` A list of values to be tried for the first `userfun` argument. This normally includes some "random" integers and some special values for which you want to be sure your program works, such as 0 and the maximum negative number. However, you might have a function that has to work for only a few values, in which case you should list only those. Execution time is not very sensitive to the length of this list; use as many values as you want. Aha! does NOT augment this list with internally generated random integers.

`trialy` A list of values to be tried for the second `userfun` argument.

`IMMEDS` A list of immediate values to be tried for all instruction operands except shift amounts. The first three values must be 0, –1, and 1, in that order. Most instructions skip the value 0, and some skip –1 and 1 (e.g., we never add 0 or multiply by 1, and so on) Execution time is very sensitive to the length of this list. If you have reason to think that certain immediate values might be used in a solution, then include those. The maximum negative number is often a good one to include. Other than those, you might include no numbers at all, or

just a few such as ±2. This list is a sequence of integers separated by commas, but with no braces around them.

`SHIMMEDS` A list of the immediate values to be tried for shift amounts. The values should range from 1 to 31. Execution time is quite sensitive to the length of this list. If you have reason to think that certain values might be used in a solution, then include those. Otherwise, 1 and 31, and possibly 2 and 30, are reasonable values to include. This list is a sequence of integers separated by commas, but with no braces around them.

Functions You must provide a simulator routine for each instruction in the target machine's ISA. See `abs.h` or `avg.h` for examples.

Instruction Definitions You must give various characteristics of each instruction. A few examples are given above on page 2, and there are many more examples in `abs.h` and `avg.h`.

In the instruction definitions, the `opndstart` entries need more explanation. These values are the first (lowest) register numbers ever used by the operand. Each value is independent of the others. To fully understand their meaning, you have to understand the way registers are used by Aha!; this is described on page 8. The essence is that you set these values as follows:

1. If the operand can be an ordinary immediate value (`IMMEDS`), use the index of the first immediate value that makes sense for the instruction. Specify 0 to use all values starting with 0, 1 to use all values starting with –1, 2 to use all values starting with 1, and so on (recall that `IMMEDS` must start with the values 0, –1, and 1, in that order).
2. If the operand is a shift amount, use `NIM`.
3. If the operand can only be a register (that holds a variable), use `RX`.

The *conditional move* instructions, such as those found on the Compac Alpha, have a target register that is conditionally set. It is both an input and an output register. You can emulate this fairly closely with a *select* instruction, in which the target register is always set. The *select* instructions have three source operands, for example

```
sellt rt,ra,rb,rc
```

tests the contents of register `ra`, and if less than 0, it copies `rb` to `rt`, and otherwise it copies `rc` to `rt`. Examples are in `abs.h`.

Because 3-operand instructions have a large number of combinations of their operands, their use is expensive in execution time.

If you get a stackdump with the first line:

```
Exception: STATUS_ACCESS_VIOLATION at eip=00000000
```

a likely reason is that you forgot to include a simulator routine for one (or more) of the instructions in the ISA, and hence the program did a branch-and-link to location 0.

Compiling, Linking, and Executing

When you are through defining your machine, you compile and link all the programs with a command such as

`make MyProblem`, on Windows, or
`make EXAMPLE=MyProblem`, on Linux.

The file `MyProblem.h` will be included in file `aha.c`. (This feature of having the file to be included defined on the command line might be difficult to implement on some platforms. If so, you can change a line in `aha.c` from `#include INC` to `#include "aha.h"` and work by always copying the problem description file you're interested in to `aha.h`).

The `make` command above creates file `MyProblem.exe`, which you run with the command

`MyProblem` *n*

where *n* is length of the programs to be tried (from 1 to 4, or 5 if you are willing to wait for a very long time for the run to end).

The solutions (if any) are displayed as they are found. They are also written to a file `MyProblem.out`.

Testing and Debugging

If you make very many changes to `abs.h` or `avg.h`, you should test them. Set the `debug` switch to 1, and execute Aha! with a small parameter such as 1 or 2. Examine the contents of the output file (e.g., `MyProblem.out`) to assure yourself that it's reasonable.

The output file contains a trace of each trial program executed and shows the input values in RX and, if applicable, RY. The result computed by each instruction executed is shown. Below is an example.

```
Simulating with trial arg x = 3 (0x3):
   not   r1,rx     ==> -4 (0xFFFFFFFC)
   add   r2,rx,1     ==> 4 (0x4)
   shl   r3,r2,r1     ==> 0 (0x0)
   Expr: ((x + 1) << ~(x))
Computed result = 0, correct result = 3, fail

Simulating with trial arg x = 3 (0x3):
   not   r1,rx     ==> -4 (0xFFFFFFFC)
   add   r2,rx,1     ==> 4 (0x4)
   shl   r3,r1,r2     ==> -64 (0xFFFFFFC0)
   Expr: (~(x) << (x + 1))
```

```
Computed result = -64, correct result = 3, fail
```

The word `fail` means the result of the simulation did not match the precomputed result for the input argument used (3). If it reports `ok` here, the next group of lines will show the simulation of the same program with a different input value.

Notice that after the *shift left* with input operands `r2,r1`, the next combination is `r1,r2` rather than an immediate shift amount followed by `r2`. This is because the `shl` is the last instruction, and `r1` has not yet been used, so the `shl` must use `r1` (and also `r2`). Due to a shortcoming in the logic of the program, the next trial program uses an `shl` with operands `r2,r2`; if the program had a little more smarts this would be skipped. (The program will sometimes simulate an unnecessary program, but it will not skip a program that should be tried.)

3 Internals

Register usage

All instruction operands are register numbers; Aha! does not deal with instruction formats that have immediate operands. Instead, the general purpose registers are used to hold both immediate values and the results of evaluating instructions. The registers are grouped in the following order, starting with register 0:

1. Immediate values (user-defined) for most instructions (`IMMEDS`)
2. Immediate values (user-defined) for the shift instructions (`SHIMMEDS`)
3. First argument of the user-defined function
4. Second argument of the user-defined function (optional)
5. Result of trial instruction 0
6. Result of trial instruction 1
7. …
8. Result of last trial instruction

Table 3–1 shows the expressions used in the code to refer to these groups of registers. Most are macro variables, so they become constants when the program is compiled. The basic variables upon which others are defined are:

`NIM`, the number of immediate values
`NSHIM`, the number of shift immediate values
`NARGS`, the number of arguments of the user-defined function (1 or 2)

The immediate values are preset into the array and they never change. The first and second arguments to `userfun` come from arrays `trialx` and `trialy`. The remaining values are computed by the instructions of the trial programs.

TABLE 3–1. REGISTER USAGE

Description	*Register Number*	*Expression*
First ordinary immediate value	`0`	`0`
...		
First shift immediate value	`NIM`	`NIM`
...		
First argument	`NIM + NSHIM`	`RX`
Second argument (optional)	`NIM + NSHIM + 1`	`RY`
Result of instruction 0	`NIM + NSHIM + NARGS`	`RI0`
...		
Result of instruction *i*	`NIM + NSHIM + NARGS + i`	`RI0 + i`

Instruction operands

There are three types of operands:

1. An operand that starts with an ordinary (non-shift) immediate value (from `IMMEDS`), continues through to the last immediate value, skips the shift immediate values, and continues to the registers that contain variables (`RX` up to the register set by the previous instruction).
2. An operand that starts with a shift immediate value (from `SHIMMEDS`) and continues to the last valid register that holds a variable.
3. An operand that cannot be immediate; it starts at `RX` and continues to the last valid register that holds a variable.

The first type of operand is the most common. It is denoted by a starting value (in `opndstart`) of 0, 1, 2, or 3. If 0, the operand runs through all the ordinary immediate values. If 1, it starts with `IMMEDS[1]`, bypassing `IMMEDS[0]`, which always contains 0, and so on.

The second type of operand is denoted by a starting register number in the range of shift immediate values (`NIM` through `NIM+NSHIM-1`); ordinarily the starting register number is `NIM`.

The third type of operand can only be a register (that contains a variable), and it is denoted by a starting value in the range of registers that contain variables, ordinarily `RX`. This type of operand is used only for the first operand of a commutative operation, which in Aha! means an instruction for which the first two operands commute. Aha! doesn't generate trial instructions in which the operands are all immediate values (as this would just generate another constant, which is assumed to be a waste of time). For a two-operand commutative operation, it puts the immediate operand, if any, second. Thus the first operand is always a variable. So far there are no three-operand commutative instructions, but the program will handle them.

As an example, for the divide instruction, `opndstart` is given as `{1,3,0}`. This means that the first operand starts with register 1, which contains

the value –1. Having a first operand (dividend) of 0 would be a waste of time, so it is bypassed. The second operand starts with register 3; thus dividing by immediate 0, –1, and 1 are bypassed (–1 is bypassed because it is presumed that the *negate* instruction is present). The last integer in `opndstart` is not used, because the *divide* instruction has only two operands.

Although 1 and 3 are, independently, the lowest values used for the *divide* instruction's operands, it is not executed with these two values for its operands. When Aha! sets up the *divide* instruction in a trial program, it initially sets the operands to register numbers 1 and 3, but then immediately invokes function `fix_operands` to "fix up" the operands so they observe three rules: (1) if the instruction is the last in the trial program, at least one of the operands must reference the immediately preceding instruction (otherwise the immediately preceding instruction would be useless), (2) if the operation is commutative, the first operand is greater than or equal to the second (as a register number), and (3) not all operands are immediate values.

It does this by "incrementing" the operands by a minimal amount so that the conditions are satisfied. When Aha! cycles through the combinations of registers for an instruction, it increments the leftmost operand until it reaches its maximum value, then it resets that operand to its minimum permitted value (defined by `opndstart`) and increments the operand to the right, and so on, much like the numbers on an odometer except in reverse order. (Certain details in the program, such as handling commutative operations, are simpler if this counter-intuitive order is used.) As an example, assume that a divide instruction is the second of three instructions, and the last immediate value is at index 4 (that is, `NIM` = 5). Then the operands get incremented in the order shown below, reading downwards. Here `R0` holds the result of the first instruction (instruction 0). The order of the register numbers is `1`, `2`, `3`, `4`, `RX`, `R0`.

```
RX,3    1,RX    RX,RX    3,R0
R0,3    2,RX    R0,RX    4,R0
RX,4    3,RX    1,R0     RX,R0
R0,4    4,RX    2,R0     R0,R0
```

If you want the *divide* instruction to cycle through only registers that contain variables, set `opndstart` to `{RX,RX,0}`.

As another example, consider the *add* instruction. For this, `opndstart` is given as `{RX,2,0}`. This means that the initial value for the first operand is register `RX`. This operand is never an immediate value, because the operation is commutative. The starting value of the second operand is 2, referring to the number two (the third) immediate value, which is 1. Since the first three immediate values are 0, –1, and 1, this specifies that the values of 0 and –1 are to be skipped, and an *add* of 1 is the first value to be considered. An *add* of –1 is skipped because presumably it will be covered by a *subtract* of 1. Assuming the *add* instruction is the second of three or more instructions, its operands cycle through the following values:

```
RX,2    RX,4    R0,R0
R0,2    R0,4
RX,3    RX,RX
R0,3    R0,RX
```

Notice that `RX,R0` is not tried; for commutative operands the rule is that the first register number must be greater than or equal to the second.

Overall Method

The number of instructions in the trial program, *n*, is given as an argument to Aha!. First the program evaluates `userfun` for all the values in `trialx` and, optionally, in `trialy`, and saves the results in a table (`correct_result`). Function `userfun` will not be evaluated anymore.

Next the program sets up the instruction array `pgm` with *n* copies of the first instruction in the ISA, and with the lowest register numbers used by that instruction (from `opndstart`). For each of these instructions, `fix_operands` is called, so their operands are copacetic.

That completes initialization, and next function `search` is called to search for solutions.

The search begins by simulating the program using the first values in array `trialx` and, optionally, `trialy`, which are placed in `RX` and `RY`. The simulation is done by function `check`. The result calculated by instruction *i* is placed in the register array at `r[i+RI0]`. The value placed in the last register is compared to the precalculated result in array `correct_result`. If the comparison result is "not equal," then `check` immediately returns to `search` with a failure indication. If the comparison result is "equal," then `check` simulates the complete program (all *n* instructions) with the next value in `trialx` and, optionally, `trialy`. For each simulation, the final result is compared to the appropriate value in array `correct_result`, and if the comparison result is "not equal," control returns with a failure indication. If all comparisons results are "equal," `check` returns with a success indication.

If a comparison result is "not equal," `check` remembers the indexes in `trialx` and `trialy`, and it will start with these values the next time it is called. This is an attempt to reduce the number of simulations required by sticking with a "good" value (one that causes failure) when one is found.

Function `check` is called with the index *i* of the lowest numbered instruction that was altered by the caller (`search`). Only instructions from *i* on to the end of the trial program are simulated. This greatly reduces the number of instructions to simulate, as usually *i* indexes the last instruction in the program.

If variable `counters` is 1, function `check` increments `counter[i]` each time it simulates the instruction at position *i*. At the end of the run the *n* counters are displayed. This is primarily of interest to the program maintainer.

Back in program `search`, if `check` returns with a success indication, the program is displayed. Then, whether success or failure occurred, the program is "incremented" and tried again. The incrementation, done by function `incre-`

`ment`, increments the leftmost operand of the last instruction in the program, if possible. If not, it resets that operand and increments the next operand to the right, and so on. If the operands cannot be increased anymore, it replaces the last instruction in the trial program with the next instruction in the ISA. If it has reached the last instruction in the ISA, it backs up and increases the previous instruction, and so on.

4 Shortcomings

Immediate operands are the bane of this program. Most RISC computers have many instructions with 16-bit immediate fields, but to allow all 65,536 values is out of the question. As soon as a program with three immediate values is tried, it would require over 2^{48} instruction simulations, which would take a few years of running time. The GNU superoptimizer allows five immediate values: 0, ±1, the maximum negative number, and the maximum positive number. Aha! allows more, but if you don't have any reason to suspect that certain immediate values might be useful, you might as well hold the list down to about those five. Thus a shortcoming of Aha! is that it will not necessarily find an *n*-instruction solution when in fact one exists.

Another is that a solution it finds may not be correct. All that can be said is that it works for the trial values in `trialx` and `trialy`, but the user must examine an alleged solution to see if it works for all input arguments.

Of course a major shortcoming is that it takes so long to execute that it can't very well be used for trial program lengths of five or more. Well, for some simplified machines, if you're willing to run the program all day, maybe it can handle five. But not six. It is entertaining to try to speed it up by tinkering with the code, but many of the things the author has tried resulted in only a 5 or 10 percent improvement, and made the program much less transparent, and so were rejected.

The maximum number of arguments to `userfun` is two. The maximum number of source operands on an instruction is three. Caution: three-operand instructions increase the execution time substantially.

Aha! does not handle the carry bit or other bits that a machine may set as a side effect of executing certain instructions. It probably should be improved to handle the carry bit, because most machines have it in some form, and both Masselin's and GNU's superoptimizers have found several interesting and unexpected problem solutions that use it.

The program does not handle two-address instructions. However, Aha! is probably useful for experimenting with such machines anyway, by fixing up the discovered code manually.

The program does not handle operands that can *only* be immediate values.

The program does not handle the possibility that register 0 is a permanent 0. However, you can fake this by adding a few instructions. For example, to get the effect of a *subtract* in which the first operand is 0, add a *negate* instruction (which has just one operand).

The program does not handle register pairs. For example, many machines have a multiply instruction that produces a two-register result, and a divide instruction for which the dividend is a register pair. These instructions cannot be handled by Aha!.

The fact that Aha! doesn't handle loads and stores can cause it to miss certain solutions. For example, suppose `userfun` has two arguments *x* and *y*, and the desired function is to transfer the rightmost byte from *y* to the rightmost byte of *x*. Then the shortest program would *store byte* from *y* and *insert byte* into *x*, assuming the machine has those instructions. But Aha! would not find that solution. This seems like a rather unimportant shortcoming.

The program also does not handle branches. And it does not handle floating-point instructions or memory-to-memory operations such as are often found on CISC's. Actually, Aha! probably could handle single-precision floating-point operations by treating the data as integers, assuming it is satisfactory to have the floating-point data reside in the same set of registers as is used by the integer and logical instructions.

5 Future Work

Find a better (simpler and faster) way to increment the operands. Table assist?

Is there a good way to ensure that all computed values are used? E.g., for `numi` = 3, the present program might generate:

```
op1   r1,rx,rx
op2   r2,rx,rx
op3   r3,rx,r2
```

which leaves `r1` unused. In such a case Aha! would have found a solution with fewer than *n* instructions, which the user presumably tried and found no such solutions. But still, if such cases can be efficiently skipped, the program would run faster.

An investigation of this showed that for a typical RISC instruction set, 39% of three-instruction programs had an unused result, and 70% of four-instruction programs had an unused result. This is compared to the program which ensures only that the second from last computed result does not go unused. Thus there is hay to be made here.

An attempt to skip *all* these silly programs resulted in a net increase in execution time, because it was implemented inefficiently. Then, as a compromise, steps were added to the program to ensure only that the second and third from last results are both used. This improved execution time by a factor of 1.8 (over a version of the program that ensured only that the second from last result is used) in the case of searching for a four-instruction solution, and the factor would probably be larger for five-instruction programs. The logic for even this simple optimization is a bit complicated. It is explained in the program.

Is there a good way to eliminate silly instructions such as subtracting a register from itself, or adding a register to itself given that we probably would have *shift left 1*, etc.?

Similarly, is there a good way to eliminate useless combinations, such as a *negate* feeding another *negate*?

The best way to find the shortest path from one node to another in a graph is to search from both ends alternately until the set of nodes reached from one end intersected with the set of nodes reached from the other end is non-null. Is there some way to solve the shortest program problem by searching from both ends? For example, assume that an *add* instruction is being tried as the last instruction. Then it is pointless to try all the immediate values. One can simply subtract the result of instruction $n-1$ from the known result and use that as the immediate value. Then, see if the program works for all the trial values. Similarly, if the last instruction is *or immediate*, then in most cases only a few immediate values will work. This would not only speed up the program considerably, but would allow an arbitrary immediate value at least in this one position.

Simulate the carry bit. This might be expensive, requiring four *add* and four *subtract* ops, rather than the present one each. To hold down the execution time cost and reduce the number of silly solutions that are printed, the program should not use the carry bit until it has been set, and should not set it twice with no use between. And the last instruction should not set it. And no program should generate the carry but never use it. How should it be shown in the expression representation of each solution? I suppose we could just have some more "+" operators, such as +gc, +ci, and +igc, but it wouldn't be clear (in general) to which operation a carry input is tied. A little gross and pretty ugly.

6 References

[GK] Torbjörn Granlund and Richard Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler." In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), July 1992, 341-352.

[HM] Henry Massalin, "Superoptimizer -- A Look at the Smallest Program." In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS II), 1987, 122-126.